ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Laurea in Ingegneria Informatica

Exploring Phase-Change Memory as a Compute Accelerator Module:
A Virtual Prototype on the PULP Platform

Relatore:
Chiar.mo Prof.
Giuseppe Tagliavini

Presentata da:
Leonardo Domenicali

Invernale
2024/2025

Exploring Phase-Change Memory as a Compute Accelerator Module:

A Virtual Prototype on the PULP Platform

Leonardo Domenicali

Abstract

Traditional von Neumann architectures are hitting a wall. The physical separation between memory and processing creates a bottleneck that is becoming critical for data-intensive applications, particularly in AI and Machine Learning. Moving data back and forth between CPU and memory is becoming more time and energy consuming than the actual computation: this is the "memory wall" problem.

This dissertation explores Phase-Change Memory (PCM) as a potential solution through Analog In-Memory Computing (AIMC). The core idea is to perform the computation directly where the data is stored instead of relying on costly moving cycles. PCM is particularly interesting for this kind of application because its analog resistance states can naturally represent neural network weights, and its crossbar architecture allows for parallel computation of matrix-vector multiplications in a single analog operation.

The main contribution of this work is a working virtual prototype of a PCM-based AIMC accelerator. The model is implemented in C++ and integrated into the GVSoC simulation platform within the PULP ecosystem. A significant part of the work focused on optimising the simulation itself, making it sufficiently fast to be useful for experimentation. This involved designing cache-friendly data-structures and implementing multithreaded execution for the core Matrix-Vector Multiplication algorithm. Performance benchmarks demonstrate the effectiveness of these optimisations. The flat buffer data-structure reduces cache misses by up to x times compared to a standard multidimensional array approach, and multithreading provides significant speedup on multi-core systems, with optimal performance achieved at 8 threads for the tested configurations. The result is a configurable simulation framework that can be used to explore AIMC architectures and provides a foundation for co-designing hardware accelerators and the software needed to use it effectively.

Acknowledgements

TODO: Write acknowledgements.

Contents

List of Figures

List of Tables

List of Abbreviations

MVM	**M**atrix **V**ector **M**ultiplication
MM	**M**atrix **M**ultiplication
NVM	**N**on-**V**olatile **M**emory
PCM	**P**hase-**C**hang **M**emory
AIMC	**A**nalog-**I**n **M**emory **C**omputing
CIM	**C**omputing **I**n-**M**emory
CNM	**C**omputing **N**ear-**M**emory
AI	**A**rtificial **I**ntelligence
ML	**M**achine **L**earning
SoC	**S**ystem **o**n **C**hip
ADC	**A**nalog to **D**igital **C**onverter
DAC	**D**igital to **A**nalog **C**onverter

1

Introduction

1.1 Von Neumann Architecture and the Memory Wall

The field of computing is at a critical point. For decades, the industry has relied on the von Neumann architecture, which physically separates processing and memory units. While incredibly successful, this paradigm has created an inherent "memory wall" [Gho+24], where the time and energy spent moving data between the CPU and memory now dominate the cost of computation. This bottleneck is especially highlighted in data-intensive fields like artificial intelligence and machine learning, domains heavily reliant on massive parallel operations such as Matrix-Vector Multiplication (MVM) and Matrix-Matrix Multiplication (MM) [Gho+24; Kha+24]. As Moore's Law slows, only shrinking transistors is no longer a viable path to performance gains, creating the need for new computing architectures.

1.2 Dissertation Objective and Motivation

The "memory wall" has motivated a global research effort into new, beyond von Neumann, computing paradigms. Among the most promising is Computing Near-Memory (CNM) and Computing In-Memory (CIM) [1.1], which idea is to perform computation directly where data is stored, eliminating the costly data movement [Kha+24]. Among the CIM branches, Phase-Change Memory (PCM) has emerged as a possible candidate to enable Analog In-Memory Computing (AIMC) as its analog resistance states can naturally represent the weights of a neural network and the way the crossbar is configured allows for efficient analog and parallel computation.

However, designing and fabricating new hardware accelerators is a high-risk, high-capital process. Before committing to silicon, it is essential to have robust virtual platforms to explore architectural trade-offs, validate functionality, and co-design the necessary

software. This project is motivated by the need for such virtual prototype.

The primary objective of this dissertation is to implement, and validate a virtual prototype of a PCM-based AIMC accelerator within the GVSoC [Bru+21] simulator for the PULP platform.

To achieve this objective two main goals were established:
Develop a C++ model of a PCM accelerator capable of performing MVM operations, integrated as a memory-mapped peripheral in GVSoC.
Investigate and implement performance optimisations for the simulation model itself to allow for efficient and fast simulation even on limited hardware resources.

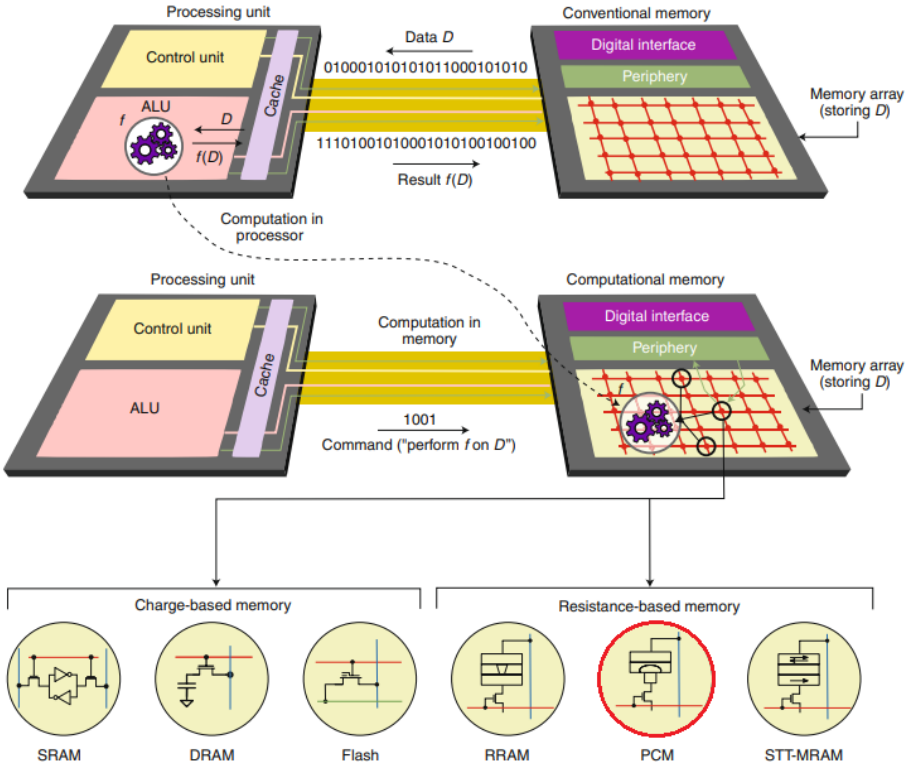


Figure 1.1: Computing In-Memory (CIM) architecture overview, moving computation closer to memory to alleviate the von Neumann bottleneck. Adapted from [Seb+20].

1.3 Summary of Contributions

This dissertation makes several contributions to the field of hardware acceleration and virtual prototyping for next-generation computing architectures. The primary outcomes of

this research are the following:
A configurable virtual prototype of a PCM accelerator. The core contribution is a fully functional C++ model of a PCM-based AIMC accelerator integrated into the GVSoC framework. This model is not just a theoretical construct but a working, configurable, peripheral that can be simulated within a complete System-on-Chip environment, serving as a valuable tool for architectural exploration.

Demonstration and analysis of simulation optimisation techniques. This work demonstrates the effectiveness of applying advanced software optimisation principles to the task of hardware simulation. By implementing and benchmarking techniques such as flattened data buffers and multithreading, this project provides concrete data on how to significantly improve the performance and scalability of virtual prototyping, making large-scale experiments more feasible even on limited computational resources.

2

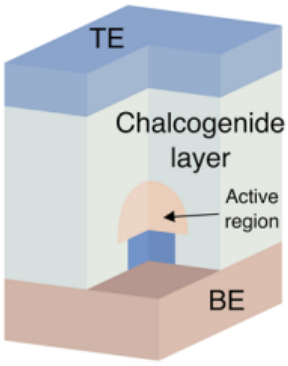
Background

This chapter introduces the main concepts and technologies that are the base of this dissertation.
We will start by introducing the Phase Change Memory (PCM), a non-volatile memory (NVM) capable of storing data by changing the state of the material, and why it is suitable for this kind of application. Then we will introduce the concept of Analog In-Memory Computing (AIMC) and how it can be used to speed up matrix-vector multiplication.
Finally we will introduce the PULP platform and the GVSoC simulator as the tools and frameworks for the module implementation.

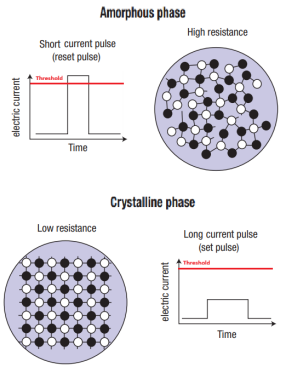
2.1 Phase Change Memory (PCM)

Phase Change Memory (PCM) [2.1a] is a type of NVM that uses the unique behaviour of some chalcogenide compound materials to switch between amorphous and crystalline states, these two phases offer differential electrical impedance that manifests as variable resistance [GS20; He+23]. By applying a current to the material we can measure the output voltage and, by using Ohm's law, we can determine the resistance of the material and therefore the value of the cell. The state change is achieved by applying different heat levels to the material using electrical pulses [2.1b]; these pulses differ in both intensity and duration depending on the desired state. For this exact reason we can notice a difference in write and read speed, with writes being slower and of variable durations but allowing reading speed comparable to DRAM read speed, or of around 50 ns [Sri+13].

By reviewing the literature we can find papers starting to describe this innovation as early as 1960s however in the following decade the interest declined until the 2000s as drift related issues emerged. In the early 2000 the technology started to be developed for commercial use, one widely recognized implementation is the Intel Optane, a PCM based technologies called 3D XPoint [GS20; He+23]. Initially conceived as a binary memory technology, PCM has evolved to support multi-level cell architectures by using multiple levels of resistance to rappresent the stored data [Ant+24]. This property makes PCM particularly suitable for MVM and MM operations, as the weights of the matrices can be stored as resistance values directly within the memory cells.



(a) PCM Cell Schematics. Adapted from [Man+23]



(b) PCM Set and Reset pulses. Adapted from [WY07]

Figure 2.1: PCM Cell Schematic and Set/Reset Pulses.

2.2 Analog In-Memory Computing (AIMC)

With Analog In-Memory Computing (AIMC) we refer to a computing paradigm that aims to perform computations directly within the memory array, thus reducing the need for data movement between memory and processing units. This approach is particularly beneficial for operations that involve large amounts of data, such as matrix-vector multiplications (MVMs) and matrix-matrix multiplications (MMs), which are common in machine learning, AI and scientific computing applications.
AIMC leverages the physical properties of memory devices, such as resistive switching in PCM, to perform computations in an analog manner. This allows for parallel processing of multiple operations by design, leading to significant speedups compared to traditional digital computing approaches. Moreover, analog capabilities scales better than digital operations leading to improved energy efficiency as the data grows.

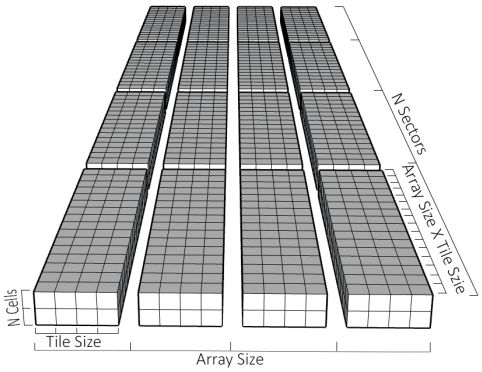
2.3 Applying AIMC to PCM

The PCM technology is particularly interesting for AIMC because it allows parallel execution of multiple operations. In fact, the internal structure of PCM crossbar-array [2.2] allows to perform matrix-vector multiplications (MVMs) in a single step: by applying the input vector as current X_i to the rows of the PCM array and reading the output current from the columns, thanks to Kirchhoff's law, we retrieve the result $Y_i = \sum_{k=0}^{n} X_i \cdot R_{ik}$ in a single operation [He+23].
Moreover, the AIMC capability of a PCM crossbar make good use of the reading operation, which is significantly faster than the writing operation optimising the overall performance for such computations. To apply this concept to a real PCM module we need to consider some additional aspects such as the presence of Digital-to-Analog Converters (DACs) to convert the digital input vector to analog signals for the rows and Analog-to-Digital Converters (ADCs) to convert the analog output currents from the columns back to digital values.

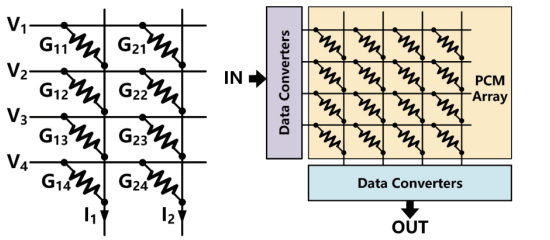
2.3.1 PCM Module Architecture

After considering the PCM cell architecture and its internal schematics we can now analyse the structure of a PCM module that can be used for AIMC. The PCM module architecture is shown in figure [2.2a].
The module is characterised by a matrix $R \times C$ of tiles. Each tile is a 3D array itself with surface $X \times Y$ and depth K, with K being the number of layers of PCM cells stacked

on top of each other. Each cell is able to holds n-bits as variable resistance. The core concept of the entire module is to hold up to L layers, as $L = C \cdot K$, stored as $X \times J$ matrices, with being $J = Y \cdot R$. During a computations is possible to select layers and matrices portions, called sectors, to use by providing the module with specific commands on the data bus. The tiles are interfaced with a set of DACs connected to the rows and ADCs connected to the columns, enabling bidirectional conversion of input vectors from digital to analog and output vectors from analog to digital, respectively. This is the basic architecture of the PCM module that we will use for our implementation.



(a) PCM Memory Architecture



(b) PCM Internal Schematics. Adapted from [He+23]

Figure 2.2:
PCM Module Architecture and Structure.

2.4 PULP Platform

Parallel Ultra Low Power (PULP) is an open-source computing platform developed by the PULP team at the University of Bologna and the ETH Zurich.

2.4.1 PULP Overview

The platform's objective is to provide high performance and energy efficient computing solutions based on the RISC-V architecture. Within the platform we can find a variety of processing units, including RISC-V cores and specialized accelerators. The framework is highly configurable, allowing designers to tailor the architecture to specific application requirements. PULP is particularly well-suited for edge computing applications, where

power consumption and performance are critical factors.

2.4.2 GVSoC Simulator

The PULP ecosystem is supported by a complete software stack, including the SoC simulator (GVSoC) [Bru+21], which provides an event-driven simulation environment for PULP-based systems. This simulator allows developers to model and simulate the behavior of systems on chip, enabling them to evaluate performance, power consumption, and other metrics before deploying on actual hardware. GVSoC supports a wide range of features, including multi-core processing, SoC clusters, memory hierarchies, and peripheral devices. Within GVSoC, developers can create virtual prototypes of custom hardware modules, such as the PCM module described in this dissertation, and integrate them into the PULP platform for simulation and testing. We'll cover this in the upcoming chapters.

3

PCM Module Implementation

This chapter will focus on the implementation of the PCM module within the GVSoC framework. We will start by introducing the architecture of the module and the data-structures used to store the crossbar values. Then we will analyse the MVM algorithm and how we can optimise it to achieve better performance.
The first step before starting the implementation is to define the constraints of the architecture and the software running the simulator.

3.1 Architecture Constraints

Before analysing the module itself it's important to define some hardware constraints of the hardware running the simulator and software constraints of the simulator itself.

3.1.1 Hardware Constraints

The hardware running the simulator spans across various architectures and configurations from low power laptops to HPC nodes and high power workstations. This is a major concern when writing optimised code, especially when micro-optimisation are considered. For this reason the optimisation will be maintained at a high level, avoiding architecture-specific optimisations and hardware accelerators usage.

3.1.2 Software Constraints

The GVSoC framework is a C++ based SoC event simulator. While it supports many architectures and configurations, the software is intentionally single-threaded to preserve event-accurate simulations. Clusters simulations on HPC nodes and high-performance workstations reveal that the main bottlenecks are single-thread performance and memory consumption. Therefore the optimisations will target both CPU efficiency and RAM usage.

3.2 PCM Module Architecture

The PCM module architecture is shown figure [2.2a] in the previous chapter.
To summarise, the module is characterised by a matrix of tiles. Each tile is a 3D array itself with surface $X \times Y$ and depth K, with K being the number of layers of PCM cells stacked on top of each other. Each cell is able to holds n-bits as variable resistance. The tiles are interfaced with a set of DACs connected to the rows and ADCs connected to the columns, enabling bidirectional conversion of input vectors from digital to analog and output vectors from analog to digital, respectively. This is the basic architecture of the PCM module used in this implementation.

3.3 Module Data-Structures

Based on the architecture of our module we notice the need of a large memory bank capable of accommodating all the values. To add an abstraction layer we can model only the layers and matrices sectors without major concern for the physical layer. The primary issue is to define a data-structure to store all the weight stored in the matrice, to allow for a fully configurable design dynamically allocated arrays comes across as the first choice However it's key to define a matrix-vector multiplication algorithm to immediately test whether the data-structure is effective.

$$Y_{s \cdot J+j} = \sum_{i=0}^{I-1} (\sum_{l \in L_s} M_{s,l,j,i} \cdot X_i) \tag{3.1}$$

Where:

s = sector index
j = tile row index
J = number of rows in the tile
l = layer index
i = column index
L_s = layers used in the sector
I = number of columns in the matrix
M = matrix
X = input vector
Y = output vector

The above equation describe the MVM operation performed by the PCM module. It iterates over each enabled sector s and for each row j of the tile, it computes the dot product

between the input vector X and the corresponding row of the matrix M across all enabled layers L_s. We are going to use this naive implementation to make initial benchmarks to find the best data-structure.

3.3.1 Data-structure Optimisation

The first step is to define a general data-structure that can be used to store the values of the PCM module.

4D Array

An immediate solution would be to use a 4 dimension dynamically allocated array. Benefits from this architecture would be an easy access for each cells with a standard n-dimensional matrix approach. However this structure is not optimal for major performance issues. A multidimensional array dynamically allocated array can create cache locality issues that might revolve in more cache misses than necessary slowing down the algorithm, moreover there is a high overhead in pointer allocations and deallocations during the lifetime of the model object. Another major issue, even more crucial for this use case is not optimal memory usage, not only each allocation might not be contiguous in memory, leading to fragmentation and inefficient use of the available memory but also pointer dereferencing and pointer arithmetic can lead to inefficient memory access patterns, especially for large dimensions of the matrix.

Flat Buffer

A second, more optimised, approach would be to use a "flat" n-dimensional array. This solution require a more advanced indexing computation, however it resolve in far fewer instruction for both allocation and deallocation. Implementing this solution show significant improvement in performance. Analysing a test program with valgrind cachegrind (a tool provided by the valgrind framework to simulate cache access) shows far fewer cache misses and and a fraction of memory reads. The fewer cache misses are the result of the sequentially stored data followed by the memory sequentially read, reduced memory reads occur because contiguous memory allocation eliminates pointer dereference overhead. One of the main drawback of this approach is the need to compute the index of each cell manually, however inline functions, macros and registers reuse strategies can help to reduce the complexity and the number of instructions required.

3.3.2 Differential Algorithm

Before moving to optimisations of the MVM algorithm it's important to note that the PCM module is capable of using differential weights. This means that each weight is represented by two PCM cells, one for the positive part and one for the negative part the final weight is computed as the difference between the two cells. This approach allows to maintain high precision despite the inevitable drift caused by the module physical characteristics. To accommodate this feature, without changing the data-structure or the MVM algorithm, it's possible to use two computation passes.

$$Y_{s \cdot J+j} = \sum_{i=0}^{I-1} (M_{s,h_l,j,i} - M_{s,l_l,j,i}) \cdot X_i = \sum_{i=0}^{I-1} M_{s,h_l,j,i} \cdot X_i - \sum_{i=0}^{I-1} M_{s,l_l,j,i} \cdot X_i \tag{3.2}$$

Where:

h_l = High Layer
l_l = Low Layer

This approach, while doubling the number of operations required, allows to maintain the same data-structure and MVM algorithm that we are going to optimise in the next section. This trade-off is acceptable because the differential approach is not commonly used and the optimisations for one of the two cases are mutually beneficial.

3.4 MVM Algorithm

In this section we are going to analyse and improve the MVM algorithm performances.

3.4.1 Standard Implementation

The standard algorithm to compute a MVM using this PCM module, that is already described in the previous section [3.1]: While some faster algorithms for sparse [Abb+23] and band matrix are describe in literature is notable that, in this generalised case, the matrix does not adhere to any of those rules and keeping track of the non-zero elements would require additional memory, which is not optimal for our use case. For this reason the optimisations will target the standard algorithm by working on the code and structures.

Looking at the data-structure we notice two main point. First of all the matrix is already divided in blocks (tiles) and sectors. As suggested by some studies [YSV15] granularity is one of the key for fast MVMs and MMs algorithms. Secondly our MVMs compute row by vector, rows that are stored sequentialy in our flat buffer, meaning our data already

follows cache friendly and granularity rules. This implementation already offer performance speed-ups while closely simulating the PCM module.

3.5 Optimisations

While this step is not necessary in a simulation environment it's key to run big simulations in a reasonable time even on low power devices enabling wider accessibility and usability.

3.5.1 Improved Cache-friendliness

From the analysis of the standard implementation [3.1] there is still some room for improvement in the cache friendliness. In particular the way layers and sectors are handled can be alligned to improve cache-friendliness.

$$Y_{s \cdot J+j} = \sum_{l \in L_s} (\sum_{i=0}^{I-1} M_{s,l,j,i} \cdot X_i) \tag{3.3}$$

By moving the layer to the outer loop cache locality is improved, by computing each layer completely before moving to the next one we are able to access the data in a more sequential manner, leading to fewer cache misses and data reads. This design change need to be accounted by the data-structure especially in the way layers and sectors are loaded into memory.

3.5.2 Register Reuse

Register reuse strategies are a great way to improve performance by reducing the number of memory accesses. Specificly to this case, computing the sum of each row into a register without the need to store intermediate results in the output vector eliminates unnecessary memory writes and reads. This approach lead to a significant reduction in memory accesses with the only drawback that is applicable only when the compiler is set to optimise the code. Analysing the assembly code generated for each of the case it's immediate to see why. In the non-optimised case the compiler store that same value on the stack and retrieves it at each iteration degrading the overall performances.

3.5.3 Multithreading

While cache-friendliness and register reuse are a great way to improve performance, multithreading is one of the best performance boosters in both MVMs and MMs. [YSV15;

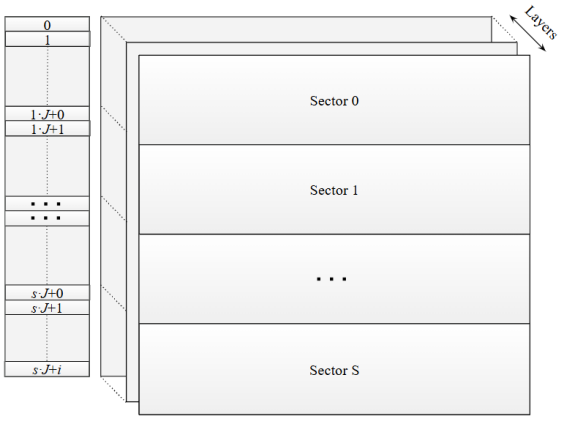


Figure 3.1: PCM matrix data-structure after cache-friendliness optimisation.

Optimised Function Entry

```
mov    r10, rdi
push   r12
lea    r9, [rcx+4]
push   rbp
mov    rbp, r8
push   rbx
mov    rbx, rdx
```

Un-Optimised Function Entry

```
push   rbp
mov    rbp, rsp
push   rbx
sub    rsp, 120
mov    QWORD PTR [rbp-88], rdi
mov    QWORD PTR [rbp-96], rsi
mov    QWORD PTR [rbp-104], rdx
mov    QWORD PTR [rbp-112], rcx
mov    QWORD PTR [rbp-120], r8
```

Optimised Variable Access

```
movsx rdx, edi
mov    r8, QWORD PTR [rbx+rdx*8]
```

Un-Optimised Variable Access

```
mov    eax, DWORD PTR [rbp-24]
cdqe
lea    rdx, [0+rax*4]
mov    rax, QWORD PTR [rbp-64]
add    rax, rdx
mov    eax, DWORD PTR [rax]
mov    DWORD PTR [rbp-68], eax
```

Table 3.1: Comparison between optimised and un-optimised function setup and variable access code.

DAm+16; Tan+24; SG98]. To both take advantage of multithreading, while minimising thread synchronisation and race conditions, that would slow down the algorithm, a modified implementation is proposed by moving some of the inner loops on different threads. The loops in questions are the sector loops. This approach guaranties granularity and, because each result value is stored at a different vector index, there is no need to synchronise processes. Moving to higher threads count than just the number of sectors is possible by splitting the row loop but require some synchronisation. In this case atomic operations

are used, the overhead is minimal because the atomic operation are limited to the write operation of the output vector however the performance impact is still present.

While implementing multithreading is fairly straight forward the problem lies in using the right amount of threads. Some studies suggest that the number of threads should be equal to the number of physical cores [BS24], using more threads will lead to "over-threading" and, therefore, worsen the performance. However others studies suggests that the number of threads is, in small variation, independent of the number of physical cores and, especially in cases where synchronisation is not required, therefore suggesting that a different number of threads could be beneficial [NK11]. For this reason algorithm tests are proposed to investigate the optimal number of threads to use in the specific case keeping in mind that the result aims to be as general as possible for consumer hardware.

3.6 Digital to Analog & Analog to Digital Converters

3.6.1 DAC

The Digital to Analog Converter (DAC) is a key component of the PCM module, responsible for converting digital input vectors into analog signals that can be applied to the rows of the PCM array. In this implementation, there is no need to simulate the actual analog signal because the PCM model, being simulated on a digital platform, can directly work with digital values.
It is important to note that the DAC introduces latency that is annotated as part of the simulation.

3.6.2 ADC

The Analog to Digital Converter (ADC) is another key component of the PCM module, responsible for converting the analog output signals from the columns of the PCM array back into digital values. Similar to the DAC, in this implementation, there's no need to simulate the actual analog signal because the PCM model can directly work with digital values, however the ADC will clip the output values to the maximum and minimum values representable by the number of bits used in the conversion. To model this behaviour, the

ADC will apply a simple clipping function to the output values.

$$n = \textit{output bit size} \tag{3.4}$$

$$ADC(y) = \begin{cases} 2^{n-1} - 1 & \text{if } y > 2^{n-1} - 1 \\ -2^{n-1} & \text{if } y < -2^{n-1} \\ y & \text{otherwise} \end{cases} \tag{3.5}$$

This function ensures that the output values are always within the valid range for the given output bit size. Similar to the DAC, the ADC introduces latency that is annotated as part of the simulation.

3.7 GVSoC Integration

To integrate the PCM module into the GVSoC framework, we need to create a new peripheral class that represents the PCM. This class will be responsible for handling the communication between the model and the rest of the system, as well as managing the internal state of the PCM. The PCM will be implemented as a memory-mapped peripheral, allowing the processor to read and write data to the PCM using standard memory access instructions. The module will be connected to the system bus, allowing it to communicate with other peripherals and memory components in the system. The bus interface will be implemented using the standard GVSoC wire.
The module have a defined mapping capable of exposing the memory space, read and write of the PCM cells, and the main AIMC unit, allowing the processor to store the input vector, trigger the MVM operation, write the input vector, and read the output vector. All the operations will expose the result immediately by queuing the result at the correct time, the only exception is the MVM operation that while taking place immediately will expose the result after a defined latency once the event-timer that signals the module. The PCM module will also include a set of configuration registers that allow the processor to configure the operation of the PCM module such as the MVM dimensions, weights precision and different MVM modes like signed and unsigned or the previously described differential mode.

4

Tests and Results

This chapter presents the results of the tests conducted and validation of the module.

4.1 Algorithm Analysis and Tests

4.1.1 Data Metrics

The project aims to provide fast and efficient MVMs for matrices stored in PCM memory up to 2 layers of 512 $\times$ 512 with 8 bit weights and 8 bit integers inputs. For this reason setups up to 512 $\times$ 512 matrices where considered. All tests were executed after a clean start-up to minimise noise given by other applications slowing down context switching and synchronization inside the CPU scheduler. The result shown where from batch runs of 100 pseudo-random matrices sequentially computed with different algorithms and different setups. By testing with this approaches, we can reduce the noise given by comparing different matrices configurations and focus on the performance of the algorithm itself. For optimisation reasons two test where conducted with optimisation switched on and off, to see how the compiler optimisations affected the results.

Performances Data

The main metric used to evaluate the performances of the algorithms is the time to complete a single MVM computed with the aid of the $std :: high_resolution_clock$.

4.1.2 Performance Analysis

The benchmark results demonstrate that the optimised implementation achieves approximately 4$\times$ speedup over the naive approach when compiled with O3 optimisations. This

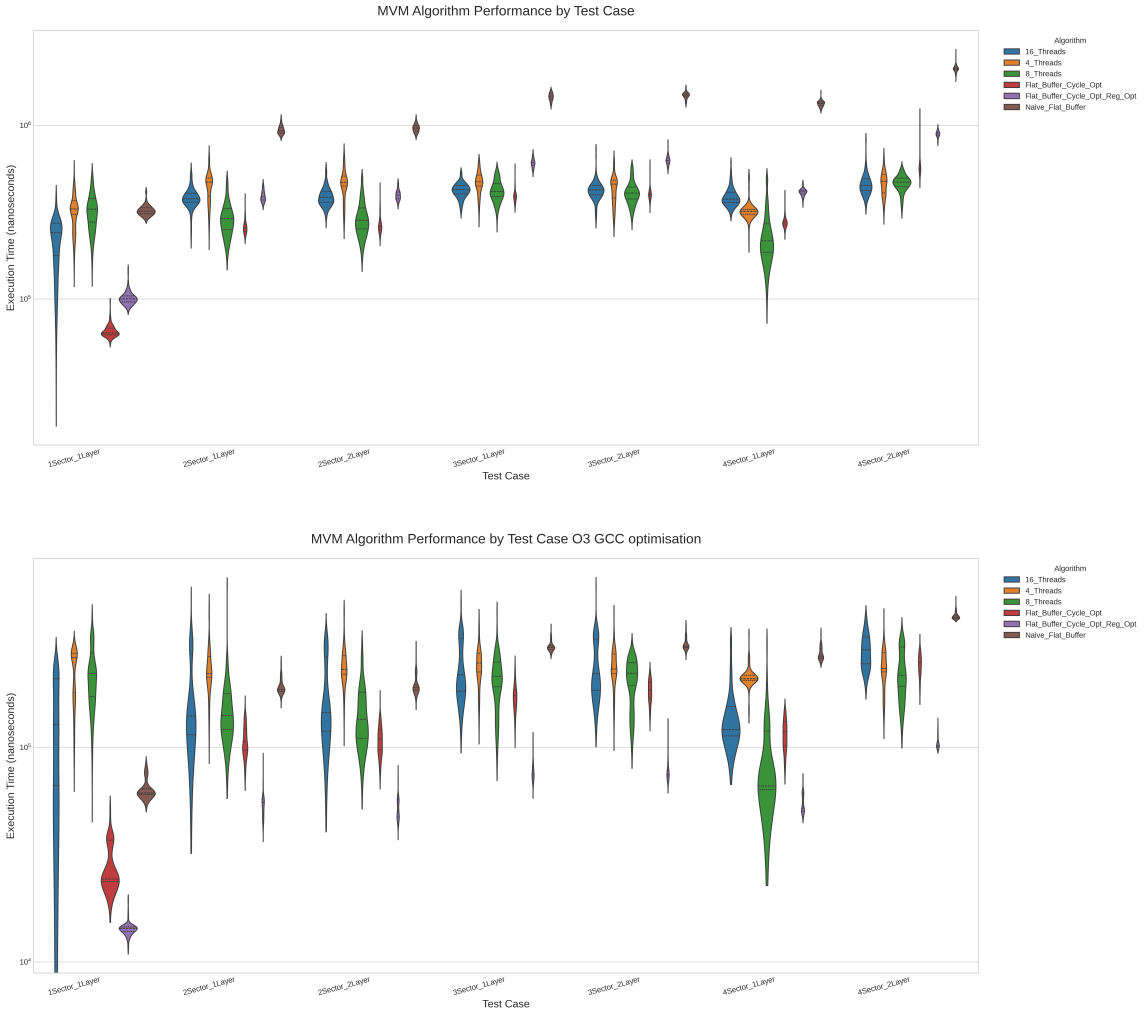


Figure 4.1:
Violin plots of the results with and without optimisation.

performance gain stems from the combined effect of algorithmic improvements and compiler optimisations working together.
The following sections analyze each optimisation technique individually to understand their contributions.

Table 4.1: Cachegrind Performance Counters for Optimised MVM Algorithms (-O3)

Algorithm	**Instructions (Ir)**	**Data Reads (Dr)**	**L1 Read Misses**	**Data Writes (Dw)**	**Branches (Bc)**	**Branch Misses**
Naive	7,343,702	1,311,265	8270	262,154	1,049,093	534
Optimise	1,814,663	66,586	8340	1027	33,809	1077
4-Thread	1,814,672	66,580	8346	1024	33,804	1071
8-Thread	1,808,472	65,544	8211	0	33,792	1050
16-Thread	1,807,632	65,552	8232	0	33,808	1066

Table 4.2: Cachegrind Performance Counters for Unoptimised MVM Algorithms (-O0)

Algorithm	**Instructions (Ir)**	**Data Reads (Dr)**	**L1 Read Misses**	**Data Writes (Dw)**	**Branches (Bc)**	**Branch Misses**
Naive	33,820,273	14,681,640	8260	1,311,250	1,049,609	555
Optimised	8,419,725	4,730,004	8335	3126	526,361	1070
4-Thread	8,419,688	4,729,980	8338	3140	526,356	1064
8-Thread	8,410,232	4,725,784	8200	5168	526,344	1055
16-Thread	8,411,440	4,726,864	8214	5248	526,352	1061

Impact of Compiler optimisations

Figure 4.1 shows the execution time distribution for different algorithm variants compiled with both O0 and O3 flags. The compiler optimisation have great impact on all implementations, with O3 producing on average 2× faster code across all tested algorithms. Under O3 optimisation, even the naive implementation benefits from compiler optimisations, though it remains the slowest one. The only exception takes place with very small matrices, where the multithreaded overhead exceeds the benefit of parallelization. Notably, compiler optimisation interacts differently with our algorithmic improvements depending on the optimisation level. Under O0, the 8-thread implementation outperforms the optimised single-threaded version, as our hand-coded optimisations cannot compensate for the lack of compiler support. However, under O3, the single-threaded optimised version becomes the fastest option, as the compiler can make good use of the optimisations we implemented through improved data-structures and memory access patterns.

Cache Behavior

Tables 4.1 and 4.2 present detailed performance counters collected using Valgrind's Cachegrind tool. These metrics provide more data on how each algorithm interacts with the CPU cache

hierarchy. Starting with the instruction counts (Ir), we observe a 4× reduction in instructions executed, this aligns with the overall memory access reductions seen in both data reads (Dr) and writes (Dw). The cache miss rate (L1 Read Misses) remains constant across all algorithms and optimisations levels, indicating that our optimisations primarily reduce the volume of memory traffic rather than improving cache hit rates. However, a closer look at the miss counts reveals that the remaining misses are the result of compulsory misses that cachegrind can not simulate away by including hardware prefetching simulation. Using the Linux perf tool, further analysis shows that the algorithms achieve an almost-zero L1-cache miss rate on typical consumer hardware. While this tool might not be as accurate as cachegrind it gives a better idea of the real cache miss rate, counting the hardware prefetcher.

Memory Access Patterns

The register reuse optimisation shows great optimisation effects on memory usage. Under O0 compilation, data reads decrease by a factor of 3.1× compared to the naive implementation, while under O3 this improvement reaches 20×. More significantly, write operations are reduced by 400× (O0) and 260× (O3). This reduction in memory traffic directly translates to performance gains, as memory accesses are often the bottleneck in MVM operations. The difference in data reads and writes between O0 and O3 highlights how compiler optimisations can further enhances the benefits of algorithmic improvements.

Multithreading Performance

The effectiveness of multithreading varies significantly based on the optimisation level. Under O0, the 8-thread implementation outperforms the single-threaded optimised version by approximately 2× on heavier workloads. This is because, without compiler optimisations, the benefits of parallel execution outweigh the overhead of thread management. However, under O3, the single-threaded optimised version becomes the fastest solution outperforming any multithreaded implementation. This shift occurs because the compiler greatly enhances the effectiveness of the single-threaded optimisations. Cachegrind results indicate that the multithreaded versions consume about the same amount of memory access as the single-threaded optimised version.

However, studies show that multithreaded implementations can incur additional memory overhead [Tan+24]. Given the memory-constrained nature of large-scale SoC simulations discussed in the previous chapter (Section 3.1.2), this overhead cannot be ignored.

Based on these findings, the final implementation uses a dynamic dispatch approach:

- When compiled with O3 optimisations: always use single-threaded optimised version

- When compiled with O0: select algorithm based on matrix setup and size
- Optional compile-time flag to force multithreading for specific usecases

To further enhance configurability, the Python generator exposes thread count as a tunable parameter, allowing users to select the number of threads based on specific workload characteristics and system capabilities.

Compiler Code Generation Analysis

Further analysis of the generated assembly code under O3 flag provides more insight into how the compiler optimisations contribute to performance improvements.

Vectorization The compiler automatically recognizes vectorizable loops and generates SIMD instructions to process multiple data elements in parallel based on the data layout and hardware resources. The specific instructions used depend on the target architecture, for the tests conducted the X86 SSE2 flags were correctly applied.

Loop Unrolling One of the most significant optimisations observed is loop unrolling. By unrolling loops, the compiler reduces the overhead of loop control instructions and increases instruction-level parallelism. The specific unrolling factor, in the tested cases, is 16 with registers of 128 bit size used to process 16 8-bit integers in parallel.

Register Allocation The compiler maintains the majority of the variables in registers during the computation, minimizing stack accesses thus reducing memory traffic.

4.2 Module Validation

To validate the module tests on the GVSoC platform were conducted and script to check the correctness of the results were implemented.

4.2.1 Algorithm Validation

To validate the algorithm two main tests were used. The first one is a simple test with a known matrix and vector, the result of the MVM is known and can be compared with the result of the algorithm. The second required randomly generated matrices and vectors, the result of the MVM is computed by the algorithm and configuration files are generated to be used in a Python script capable of computing the same MVM using NumPy as reference. The results of the two tests were compared to produce a pass or fail result. As the algorithm

is both deterministic and only uses integer arithmetic, the results are expected to be exactly the same without any variance. After thorough testing the algorithm was validated passing all tests.

4.2.2 GVSoC Integration Validation

To validate the integrations with GVSoC different tests were performed. To maintain the scope of the project the value of the weights were limited to 4 bit integers, however the module is fully parametric and can be used with any number of bits. To validate such small weights simple tests were conducted. The clipping of the weights was the first aspect to be validated, tests with weights out of range were run and the result of the MVM was checked to be correct. Then tests with randomised vector and matrix values with all ones weights were executed, the result of the MVM is known and can be easily checked as $Y_i = \sum_i^N 1 \cdot X_i$ making the expected result appear on every output value. Finally a test with random weights and vectors containing the sign of the weights was conducted, as before the result of the MVM was easily checked as the sum of the values of the rows with the sign of the weights. To further validate the module, using the same approach as before, a value within each row of the matrix was set to zero and the result was checked by comparing the difference, given by excluding the zeroed value, with the expected value. All tests passed. The module correctly handles edge cases like weight clipping and produces identical results to NumPy reference implementations.

5

Conclusion & Future Work

5.1 Conclusion

In this dissertation we designed, implemented, and validated a virtual prototype of a Phase-Change-Memory-based analog in-memory computing accelerator within the GVSoC simulation environment for the PULP platform. The core of the work is a C++ model that performs efficient matrix-vector multiplication while closely reproducing PCM device physics, including the analog resistance states and the associated read/write dynamics. This model was seamlessly integrated into GVSoC as a memory-mapped peripheral, which allows full-system simulation and enables the accelerator's performance to be evaluated within a complete system-on-chip environment. To improve simulation speed and scalability, we flattened data-structures to optimise memory-access patterns, reducing cache misses and increasing overall throughput, and we introduced multithreaded kernels that parallelise the MVM computations across cores. Extensive benchmarking identified the optimal thread counts for a variety of hardware configurations. The final accelerator model is highly configurable: users can adjust matrix sizes, cell sizes, and threading options, making the prototype a valuable tool for exploring architectural trade-offs in analog in-memory computing systems.

5.2 Future Work

While this dissertation has laid a solid foundation, there are several ways for future work that can further enhance the PCM-based AIMC accelerator model and its applications:

Extending the PCM Model: Future work could focus on incorporating more detailed physical models of PCM devices, including variability, endurance, and retention characteristics. While the current model it's already capable of simulating drift resistance MVM by using the differential algorithm a well known effect of PCM devices is the variability of read operation while under thermal stress. The current model exposes read, write and computation counters that could be used for further analysis of these effects.

Exploring Additional Optimisation Techniques to Improve Simulation Performance: Further research could investigate additional optimisation strategies. The current data-structure and the module itself could be further optimised by exploring both SIMD instructions by-design and GPU acceleration. Without the need to move the PCM weights frequently, this module is a perfect candidate for GPU acceleration.

Bibliography

[SG98] P.D. Sulatycke and K. Ghose. "Caching-efficient multithreaded fast multiplication of sparse matrices". In: *Proceedings of the First Merged International Parallel Processing Symposium and Symposium on Parallel and Distributed Processing*. ISSN: 1063-7133. Mar. 1998, pp. 117–123. DOI: 10.1109/IPPS.1998.669899. URL: https://ieeexplore.ieee.org/document/669899 (visited on 04/18/2025).

[WY07] Matthias Wuttig and Noboru Yamada. "Phase-change materials for rewriteable data storage". en. In: *Nature Mater* 6.11 (Nov. 2007). Publisher: Nature Publishing Group, pp. 824–832. ISSN: 1476-4660. DOI: 10.1038/nmat2009. URL: https://www.nature.com/articles/nmat2009 (visited on 09/12/2025).

[NK11] Alexandru Nicolau and Arun Kejariwal. "How Many Threads to Spawn during Program Multithreading?" In: *Languages and Compilers for Parallel Computing*. Ed. by Keith Cooper, John Mellor-Crummey, and Vivek Sarkar. Vol. 6548. Series Title: Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 166–183. ISBN: 978-3-642-19594-5 978-3-642-19595-2. DOI: 10.1007/978-3-642-19595-2_12. URL: http://link.springer.com/10.1007/978-3-642-19595-2_12 (visited on 04/19/2025).

[Sri+13] Rajarajan Srinivasan et al. "A Study on the Challenges and Prospects of PCM Based Main Memory Architectures". In: *Middle East Journal of Scientific Research* 18 (Dec. 2013), pp. 788–795. DOI: 10.5829/idosi.mejsr.2013.18.6.12500.

[YSV15] Dash Yajnaseni, Kumar Sanjay, and Patle V.K. *A Survey on Serial and Parallel Optimization Techniques Applicable for Matrix Multiplication Algorithm*. Tech. rep. School of Studies inComputer Science &IT, Pt. RavishankarShukla University,Raipur, Chhattisgarh,492010, India., Jan. 2015. URL: https://www.researchgate.net/publication/285131617_A_Survey_on_Serial_

and_Parallel_Optimization_Techniques_Applicable_for_Matrix_Multiplication_Algorithm (visited on 04/18/2025).

[DAm+16] Luisa D'Amore et al. "Mathematical Approach to the Performance Evaluation of Matrix Multiply Algorithm". en. In: *Parallel Processing and Applied Mathematics*. Ed. by Roman Wyrzykowski et al. Vol. 9574. Series Title: Lecture Notes in Computer Science. Cham: Springer International Publishing, 2016, pp. 25–34. ISBN: 978-3-319-32151-6 978-3-319-32152-3. DOI: 10.1007/978-3-319-32152-3_3. URL: http://link.springer.com/10.1007/978-3-319-32152-3_3 (visited on 04/18/2025).

[GS20] Manuel Le Gallo and Abu Sebastian. "An overview of phase-change memory device physics". en. In: *J. Phys. D: Appl. Phys.* 53.21 (Mar. 2020). Publisher: IOP Publishing, p. 213002. ISSN: 0022-3727. DOI: 10.1088/1361-6463/ab7794. URL: https://iopscience.iop.org/article/10.1088/1361-6463/ab7794/meta (visited on 09/12/2025).

[Seb+20] Abu Sebastian et al. "Memory devices and applications for in-memory computing". en. In: *Nat. Nanotechnol.* 15.7 (July 2020). Publisher: Nature Publishing Group, pp. 529–544. ISSN: 1748-3395. DOI: 10.1038/s41565-020-0655-z. URL: https://www.nature.com/articles/s41565-020-0655-z (visited on 09/12/2025).

[Bru+21] Nazareno Bruschi et al. "GVSoC: A Highly Configurable, Fast and Accurate Full-Platform Simulator for RISC-V based IoT Processors". In: *2021 IEEE 39th International Conference on Computer Design (ICCD)*. arXiv:2201.08166 [eess]. Oct. 2021, pp. 409–416. DOI: 10.1109/ICCD53106.2021.00071. URL: http://arxiv.org/abs/2201.08166 (visited on 09/12/2025).

[Abb+23] Amir Abboud et al. *The Time Complexity of Fully Sparse Matrix Multiplication*. Version Number: 1. 2023. DOI: 10.48550/ARXIV.2309.06317. URL: https://arxiv.org/abs/2309.06317 (visited on 04/19/2025).

[He+23] Luchang He et al. "In-memory computing based on phase change memory for high energy efficiency". en. In: *Sci. China Inf. Sci.* 66.10 (Nov. 2023), p. 200402. ISSN: 1674-733X, 1869-1919. DOI: 10.1007/s11432-023-3789-7. URL: https://link.springer.com/10.1007/s11432-023-3789-7 (visited on 04/18/2025).

[Man+23] P. Mannocci et al. "In-memory computing with emerging memory devices: Status and outlook". en. In: *APL Machine Learning* 1.1 (Mar. 2023), p. 010902. ISSN: 2770-9019. DOI: 10.1063/5.0136403. URL: https://pubs.aip.

org/aml/article/1/1/010902/2878744/In-memory-computing-with-emerging-memory-devices (visited on 09/12/2025).

[Ant+24] Alessio Antolini et al. "A Readout Scheme for PCM-Based Analog In-Memory Computing With Drift Compensation Through Reference Conductance Tracking". In: *IEEE Open Journal of the Solid-State Circuits Society* 4 (2024), pp. 69–82. ISSN: 2644-1349. DOI: 10.1109/OJSSCS.2024.3432468. URL: https://ieeexplore.ieee.org/document/10609348 (visited on 09/12/2025).

[BS24] Preet Bhutani and Amol Ashokrao Shinde. "Exploring the Impact of Multithreading on System Resource Utilization and Efficiency". In: *IJIREM* 11.5 (Oct. 2024), pp. 66–72. ISSN: 23500557. DOI: 10.55524/ijirem.2024.11.5.9. URL: https://ijirem.org/view_abstract.php?title=Exploring-the-Impact-of-Multithreading-on-System-Resource-Utilization-and-Efficiency&year=2024&vol=11&primary=QVJULTE4MjM= (visited on 04/19/2025).

[Gho+24] Amir Gholami et al. *AI and Memory Wall*. arXiv:2403.14123 [cs] version: 1. Mar. 2024. DOI: 10.48550/arXiv.2403.14123. URL: http://arxiv.org/abs/2403.14123 (visited on 09/12/2025).

[Kha+24] Asif Ali Khan et al. *The Landscape of Compute-near-memory and Compute-in-memory: A Research and Commercial Overview*. arXiv:2401.14428 [cs] version: 1. Jan. 2024. DOI: 10.48550/arXiv.2401.14428. URL: http://arxiv.org/abs/2401.14428 (visited on 09/12/2025).

[Tan+24] Tao Tang et al. "Multithreaded Reproducible Banded Matrix-Vector Multiplication". en. In: *Mathematics* 12.3 (Jan. 2024). Number: 3 Publisher: Multidisciplinary Digital Publishing Institute, p. 422. ISSN: 2227-7390. DOI: 10.3390/math12030422. URL: https://www.mdpi.com/2227-7390/12/3/422 (visited on 04/18/2025).